Replacing Training with Reasoning: Reinterpreting Classic ML Pipelines with LLMs

Marco Alecci
University of Luxembourg
Luxembourg, Luxembourg
marco.alecci@uni.lu

Jordan Samhi
University of Luxembourg
Luxembourg, Luxembourg
jordan.samhi@uni.lu

Tegawendé F. Bissyandé
University of Luxembourg
Luxembourg, Luxembourg
tegawende.bissyande@uni.lu

Jacques Klein
University of Luxembourg
Luxembourg, Luxembourg
jacques.klein@uni.lu

Abstract

Large Language Models (LLMs) are increasingly used in software engineering tasks due to their strong performance across diverse applications. In this paper, we ask a fundamental and novel question: *To what extent can LLMs replace traditional machine learning pipelines that rely on labeled data, feature engineering, and retraining?*

Our intuition is that many long-standing approaches in software engineering can be reimagined through the lens of reasoning rather than training. Unlike conventional pipelines that learn statistical patterns from data, LLMs can directly reason about contextual consistency using their pretrained knowledge. To illustrate this idea, we revisit a well-known anomaly detection pipeline (CHABADA for Android apps) and show how its clustering and retraining stages can be replaced with a simple prompting strategy. The result is a streamlined, zero-shot workflow that leverages semantic reasoning without labeled datasets, feature extraction, or retraining.

Our goal is not to propose a new tool, but to highlight a broader paradigm: LLMs open the door to reinterpreting established ML-based workflows as reasoning pipelines. This perspective suggests a path toward lighter-weight, training-free alternatives for many specialized software engineering tasks.

ACM Reference Format:
Marco Alecci, Jordan Samhi, Tegawendé F. Bissyandé, and Jacques Klein. 2026. Replacing Training with Reasoning: Reinterpreting Classic ML Pipelines with LLMs. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-NIER '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3786582.3786807

1 Introduction

Large Language Models (LLMs) are increasingly being adopted across a wide range of software engineering tasks, including bug detection and fixing [7, 12, 17, 19], testing [9, 15, 20, 29], vulnerability detection [14, 25, 32, 34], and other applications [3, 18, 21, 23, 31, 35]. Their ability to understand and reason over both natural and programming languages enables them to bridge gaps traditionally filled by domain-specific models and heuristics.

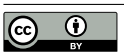
As LLM capabilities continue to advance, a key question arises: *To what extent can LLMs replace traditional machine learning pipelines that rely on labeled data, feature extraction, and retraining?* In this work, we approach this question from a conceptual standpoint: can workflows once designed around *training* be reimagined around *reasoning*?

It is important to clarify what we mean by *"reasoning"* in this context (and within the scope of this paper). While the LLM is, of course, pretrained, our method does not involve any task-specific fine-tuning, supervised learning, or labeled examples. Instead, the model draws on its general contextual knowledge to perform a task, assessing whether a given behavior is semantically consistent with its natural language description. This stands in contrast to traditional pipelines, which construct statistical models over curated datasets and must be retrained as data evolves.

We ground this exploration in a concrete case study: Android malware detection. Specifically, we revisit the CHABADA system [13], which clusters apps by their textual descriptions and applies anomaly detection to flag sensitive API usage that deviates from expected patterns. CHABADA exemplifies the costs of traditional ML pipelines—large labeled datasets, extensive feature engineering, and continuous retraining to adapt to the evolving Android ecosystem. In reimagining CHABADA, we bypass clustering and anomaly modeling entirely. Instead, we prompt an LLM—without any fine-tuning—to assess whether an app's sensitive API usage is semantically justified given its natural language description. In effect, the LLM acts as a context-aware semantic auditor, using its general software knowledge to evaluate behavioral plausibility in a zero-shot setting.

Our aim is not to propose a production-grade malware detector, nor to compete directly with state-of-the-art tools. Rather, malware detection serves here as a representative example of a broader paradigm: **established ML-based workflows can be reformulated as reasoning pipelines when powered by LLMs.** By showing how an older system like CHABADA can be simplified through this perspective, we highlight the potential of LLMs to act as flexible, training-free alternatives in specialized software engineering workflows.

Contributions. This paper makes the following contributions:



ICSE-NIER '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2425-1/26/04
https://doi.org/10.1145/3786582.3786807

- We introduce the idea of reinterpreting traditional ML-based pipelines through the lens of LLM reasoning rather than training.
- We demonstrate this perspective via a case study that reimagines the CHABADA anomaly detection system.
- We discuss the implications of this reasoning-based view for future software engineering research and practice.

Data Availability. We publicly release all associated resources to facilitate further research:

https://github.com/Trustworthy-Software/RTWR

2 Approach

Traditional machine learning pipelines for software analysis often rely on clustering, feature engineering, and anomaly detection. A representative example is CHABADA [13], which clusters apps by their textual descriptions and then models expected API usage within each cluster to flag deviations. While effective, this design exemplifies the costs of training-based approaches: large labeled datasets, complex feature extraction, and continuous retraining to adapt to evolving ecosystems.

In contrast, our approach removes both the clustering and anomaly detection stages entirely. Instead, we use an LLM that directly reasons about the semantic relationship between an app's description and its sensitive API usage. This reframing allows us to bypass training and feature engineering, replacing them with a zero-shot reasoning process.

An overview of our pipeline is shown in Figure 1. It consists of two main stages, described below.

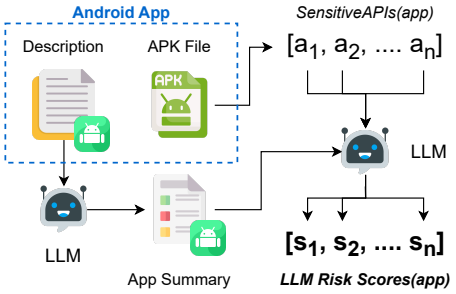


Figure 1: Approach Overview.

2.1 Sensitive API Usage Extraction

The first step involves extracting the set of sensitive API calls used by each app. We use ApkTool [6] and Androguard [5] to identify sensitive method invocations in each APK file. To ensure comparability with CHABADA, we adopt the same list of sensitive APIs used in their original study, which is based on the permission-to-method mapping introduced by Felt et al. [11]. While we acknowledge that this list may be outdated—and that more recent and comprehensive mappings exist—our focus in this exploratory work is to investigate whether LLM-based anomaly detection can replace traditional approaches compared to a known tool, such as CHABADA, in a controlled setting. Expanding the coverage to more recent mappings between APIs and permissions is left for future work. The output of this step is a list of sensitive APIs associated with each app:

$$\text{SensitiveAPIs}(\text{app}) = [a_1, a_2, \ldots, a_n]$$

where each a_i is an API e.g `getLastKnownLocation()` used by the app under analysis.

2.2 LLM-Based Anomaly Inference

Unlike CHABADA, which relies on clustering and trains a separate anomaly detector per cluster, our method analyzes each app independently. We use two prompts to guide the LLM in assessing whether each sensitive API a_i in $[a_1, a_2, \ldots, a_n]$ aligns with the app's intended functionality. In the first prompt, the LLM is asked to infer a fine-grained app category and to list the core functionalities based on the app's developer-provided description. Our hypothesis is that explicitly extracting this contextual information enables better downstream reasoning. In Section 3 (see RQ3), we validate this design choice through an ablation study.

> **[Prompt 1 Template]**
> You are an expert in Android app analysis. I will provide you with the official description of an Android app from its developer. Your task is to extract and summarize the app's key elements in a structured format.
> Instructions:
> (1) Identify the category of the app (e.g., "Social Media", "Game", "Productivity", "Finance", etc.). Be as specific as possible.
> (2) Extract the app's expected main functionalities (e.g., "Messaging", "Task Management", "Fitness Tracking").
> Return the output in the following format:
> - category: `<category>`
> - functionalities: [`<f1>`, `<f2>`, ...]

In the second prompt [1], we evaluate each extracted sensitive API a_i using the context of the app, i.e., the output of the first prompt from which the API was extracted. The LLM receives ① the app's package name, ② its summarized functionality (i.e., the output from Prompt 1), and ③ the API call under analysis. It is then instructed to assign a *context mismatch score* from 1 (fully expected) to 5 (highly suspicious). The underlying intuition mirrors that of CHABADA: we expect higher scores when an API call deviates from the app's intended behavior (e.g., a calculator app accessing location services) and lower scores when the usage is semantically aligned with the app's purpose (e.g., a navigation app accessing location).

To mitigate LLM variability and potential hallucinations, we run this prompt five times and average the results to produce a more stable score. Notably, we observed that the scores are generally very consistent across runs, with little variation (e.g., no cases like 1,1,1,1,5), showing the LLM produces stable and reliable outputs for this task. The final output for each app is thus a list of averaged context mismatch scores for its sensitive API calls:

$$\text{LLM Risk Scores}(\text{app}) = [s_1, s_2, \ldots, s_n]$$

Here, s_i denotes the average risk score for each sensitive API a_i (both vectors have the same length n), with $s_i \in [1.0, 5.0]$. In the upcoming Section 3, we will explore how these scores differ in benign and malicious apps and how they can be leveraged to perform malware detection. Importantly, this entire analysis is

[1] The prompt template can be found in our repository

performed without relying on clustering, labeled training data, or fine-tuning, allowing the method to directly analyze individual apps in a zero-shot setting. As a result, it can be readily applied to newly released or previously unseen apps, without requiring retraining or access to historical data.

3 Case Study: Revisiting CHABADA with LLMs

In this section, we describe the dataset used in our preliminary experiments and present preliminary results addressing three key research questions.

Dataset. We initially planned to evaluate our method using the original CHABADA dataset, publicly released by the authors [13], but as it includes only package names and analysis results, the app descriptions were missing. After contacting the authors without success, we explored retrieving descriptions via AndroZoo metadata [1], but since metadata collection began in 2020, most CHABADA apps from 2014 were not covered. Retrieving descriptions directly from the Google Play Store was only partially successful. We obtained descriptions for less than 10% of the original dataset, and for only 7 malicious apps, making it unsuitable for a comprehensive evaluation. Additionally, these descriptions reflect the current Play Store versions and may differ from those originally used by CHABADA, which again prevents a proper evaluation. As a result, we decided to construct a new dataset by sampling apps from AndroZoo [1, 4], which now includes metadata such as app descriptions, making it well-suited for our setting. For this preliminary study, we randomly selected 1000 benign and 1000 malicious apps from the past five years (i.e., since AndroZoo metadata collection began). To distinguish between benign and malicious samples, we followed a common practice in the literature by leveraging the VirusTotal [33] scan results provided by AndroZoo [2, 8, 10, 16, 24, 26–28, 30]. Specifically, we considered an app to be ① *Benign* if its VirusTotal score is zero—i.e., it was not flagged by any antivirus engines in VirusTotal; and ② *Malicious* if its VirusTotal score is greater than or equal to 10—i.e., it was flagged by at least 10 antivirus engines in VirusTotal.

Evaluation. With this curated dataset, our evaluation aims to answer the following research questions:

- **RQ1:** How significant is the difference in the risk scores reported by the LLM between benign and malicious apps?
- **RQ2:** How does our reasoning-based approach compare to the original CHABADA training-based pipeline?
- **RQ3:** Does incorporating both category and functionality summaries improve detection performance? What is the impact of removing this step and using only the app description?

3.1 RQ1: Differences in the risk scores.

In this research question, we evaluate whether the average risk scores assigned by the LLM, using the procedure described in Section 2, can effectively distinguish between benign and malicious apps. We employ OpenAI's `gpt-4o-mini` model [22] via the OpenAI API, chosen for its favorable trade-off between performance and cost. Given the exploratory nature of this study, we leave comparisons with other LLMs to future work. Figure 2 presents the distribution of the average LLM-assigned risk score *per app* (i.e., one average computed over the entire risk score vector for each app)

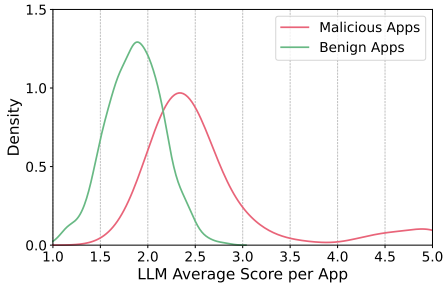


Figure 2: Distribution of LLM average score per app.

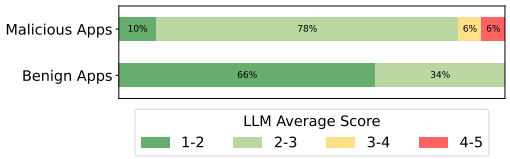


Figure 3: Distribution of LLM average scores grouped.

for both benign and malicious samples. Figure 3 complements this by showing a stacked bar plot that aggregates scores into discrete risk intervals.

As shown in Figure 2, benign apps tend to receive lower average scores, with a peak density around the 2.0 mark. In contrast, malicious apps exhibit a flatter and more right-skewed distribution, with notable densities between 3.5 and 4.5. This indicates that the LLM frequently flags their API usage as abnormal relative to their descriptions. Notably, there are no benign apps with an average score greater than 3. Figure 3 reinforces this trend by categorizing scores into four intervals. All the benign apps fall into the first two bins, while over 12% of malicious apps receive scores above 3. These patterns suggest that the LLM is effective at distinguishing between benign and malicious behavior based on semantic inconsistencies alone. For instance, among the malicious apps, the game *com.stealinggames.run* received an average score of 3.87 and a median of 4.2. Upon inspection, we found it invoked APIs such as `<TelephonyManager.getDeviceId()>` and `<LocationManager.getLastKnownLocation()>`. These calls are highly suspicious for a simple game: a high score, e.g., above 3.5, can indicate a strong likelihood of malicious behavior.

> **Answer to RQ1:** The LLM assigns significantly higher average risk scores to malicious apps than to benign ones.

3.2 RQ2: Reasoning vs. Training Pipelines

CHABADA flags an entire app as anomalous based on clustering and anomaly modeling, whereas our method produces per-API risk scores that must be aggregated. Thus, we require a strategy to aggregate these scores into a final classification for the app. A naïve heuristic would be to define a threshold for the average risk score and classify any app above it as malicious. For instance, as shown in Figure 2, setting a threshold of 3 would correctly identify all malicious apps without generating false positives (i.e., the precision

of 100%). However, this would come at the cost of a very low recall, since many malicious apps have average scores closer to 2.5.

To improve recall while maintaining high precision, we explored a more nuanced thresholding strategy. Specifically, we used the following rule: **An app is considered malicious if more than 10% of its sensitive APIs have received a score greater than 4.** Here, 10% and 4 are two parameters—denoted as X and Y—that can be tuned. In general, an app could be classified as malicious if X% of its sensitive APIs have risk scores exceeding Y. We empirically found that $X = 10$ and $Y = 4$ yielded the best results in our dataset. Given the exploratory nature of this work and the already promising results, a more thorough analysis of how to optimize these parameters is left for future work. Using this heuristic, we computed standard performance metrics such as Precision, Recall, and F_1 Score, which are reported in Table 1.

It is important to mention that to provide a baseline for comparison, we re-implemented the CHABADA pipeline using a best-effort strategy, as the authors did not release source code or provide sufficient details (e.g., OC-SVM parameters) to enable full replication. Our implementation follows everything described in the original paper to the extent possible, and the performance metrics we obtained, such as the F_1 score, are aligned with the results reported in their study (F_1 score was 0.31 in their original paper).

Table 1: Comparison of performance with CHABADA.

Approach	Precision	Recall	F_1-Score
CHABADA	0.37	0.42	0.39
This Paper	0.97	0.84	0.90

We can observe that our approach significantly outperforms CHABADA in all three metrics. In particular, it achieves very high precision (0.97), meaning it produces almost no false positives. Recall is also strong (0.84), indicating that the LLM-based approach is able to detect most of the malicious apps.

Answer to RQ2: A reasoning-based pipeline achieves substantially higher performance than CHABADA's training-based workflow on this dataset.

3.3 RQ3: Ablation Study

Finally, we investigated how the two-prompt strategy impacts detection performance. In our standard setup, the app description is first used to generate a high-level category and a summary of its intended functionalities. These are then used to assess the plausibility of sensitive API usage. To measure the contribution of this intermediate step, we repeated the experiments using a simplified version of PROMPT_TEMPLATE_2, in which the raw app description is directly provided to the LLM, without asking it to first generate a category and summary. We recomputed Precision, Recall, and F_1-Score. The results are reported in Table 2.

Removing the summarization step lowers precision from 0.97 to 0.77, while recall remains unchanged, suggesting that the category and functionality summaries help the LLM better contextualize app behavior.

Table 2: Ablation study results.

Approach	Precision	Recall	F_1-Score
This Paper	0.97	0.84	0.90
Ablated Version	0.77	0.84	0.80

Answer to RQ3: The ablation study suggests that the summarization step enhances classification precision.

4 Future Plans

This study represents an initial step in a larger research agenda. We plan to expand and deepen the reasoning-vs-training perspective by pursuing the following directions:

❶ **Broader Case Studies with Systematic Evaluation.** We plan to extend our experiments beyond Android malware detection to other software engineering fields where classic ML pipelines have been widely adopted, such as bug detection, automated testing, and program repair. The idea is to reinterpret these tasks using LLMs and compare them against their traditional ML-based versions, performing rigorous evaluations using diverse datasets with reliable ground truth. This will help establish whether reasoning pipelines generalize across software engineering tasks.

❷ **Developing Guidelines and Principles.** We aim to understand when "reasoning" pipelines should be preferred over traditional ML training pipelines. This includes identifying the types of tasks where LLMs are most effective. For example, it would be interesting to compare pipelines involving natural language artifacts, such as descriptions and documentation, with pipelines where only code is present. The goal is to provide actionable guidance to the community by clarifying the boundaries and opportunities of reasoning and LLM capabilities in software engineering tasks.

❸ **Scalability and Cost.** We will evaluate the cost and latency of reasoning-based workflows at scale and investigate optimizations such as batching, adaptive prompting, and open-source LLMs to reduce overhead.

Together, these steps will allow us to transform this exploratory idea into a systematic investigation of how LLMs can reshape software engineering workflows by replacing training with reasoning.

5 Conclusions

In this paper, we explored whether LLMs can replace traditional machine learning pipelines by reasoning directly over contextual information, rather than relying on labeled data, feature extraction, and retraining. We introduced the idea of *reinterpreting established ML-based workflows as reasoning pipelines*, using Android malware detection and the CHABADA system as a concrete case study.

Our findings suggest that a simple prompting strategy can stand in for multiple stages of a traditional pipeline, eliminating the need for clustering and anomaly modeling while still providing meaningful results. While malware detection is not our end goal, the case study illustrates how LLMs can act as flexible, training-free semantic auditors. More broadly, this perspective points toward a paradigm shift: LLMs enable software engineering research to revisit long-standing approaches and reimagine them through reasoning rather than training.

Acknowledgments

This research was funded in whole, or in part, by the Luxembourg National Research Fund (FNR), REPROCESS grant reference C21/IS/16344458.

References

[1] Marco Alecci, Pedro Jesús Ruiz Jiménez, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2024. AndroZoo: A Retrospective with a Glimpse into the Future. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 389–393.

[2] Marco Alecci, Jordan Samhi, Tegawendé F Bissyandé, and Jacques Klein. 2024. Revisiting android app categorization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[3] Marco Alecci, Nicolas Sannier, Marcello Ceci, Sallam Abualhaija, Jordan Samhi, Domenico Bianculli, Tegawendé François d Assise BISSYANDE, and Jacques Klein. 2025. Toward LLM-Driven GDPR Compliance Checking for Android Apps. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion'25)*.

[4] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 468–471. doi:10.1145/2901739.2903508

[5] Androguard. [n. d.]. Androguard: A reverse engineering tool for Android applications. https://github.com/androguard/androguard. Accessed: 2025-05-15.

[6] ApkTool. [n. d.]. ApkTool: A tool for reverse engineering Android APK files. https://ibotpeaches.github.io/Apktool/. Accessed: 2025-05-15.

[7] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134* (2024).

[8] Yizheng Chen, Zhoujie Ding, and David Wagner. 2023. Continuous learning for android malware detection. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1127–1144.

[9] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 572–576.

[10] Nadia Daoudi, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2021. Lessons learnt on reproducibility in machine learning based android malware detection. *Empirical Software Engineering* 26, 4 (2021), 74.

[11] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*. 627–638.

[12] Sidong Feng and Chunyang Chen. 2024. Prompting is all you need: Automated android bug replay with large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[13] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the 36th international conference on software engineering*. 1025–1035.

[14] Yuejun Guo, Constantinos Patsakis, Qiang Hu, Qiang Tang, and Fran Casino. 2024. Outside the comfort zone: Analysing llm capabilities in software vulnerability detection. In *European symposium on research in computer security*. Springer, 271–289.

[15] Yuchao Huang, Junjie Wang, Zhe Liu, Yawen Wang, Song Wang, Chunyang Chen, Yuanzhe Hu, and Qing Wang. 2024. Crashtranslator: Automatically reproducing mobile application crashes directly from stack trace. In *Proceedings of the 46th ieee/acm international conference on software engineering*. 1–13.

[16] Pedro Jesús Ruiz Jiménez, Jordan Samhi, Tegawendé F Bissyandé, and Jacques Klein. 2025. Dissecting APKs from Google Play: Trends, Insights and Security Implications. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 728–739.

[17] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[18] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2023. Understanding the effectiveness of large language models in detecting security vulnerabilities. *arXiv preprint arXiv:2311.16169* (2023).

[19] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499.

[20] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[21] Wei Ma, Shangqing Liu, Zhihao Lin, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, Li Li, and Yang Liu. 2023. LMs: Understanding Code Syntax and Semantics for Code Analysis. *arXiv preprint arXiv:2305.12138* (2023).

[22] OpenAI. 2024. GPT-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/ Accessed: 2025-04-24.

[23] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants?. In *International Conference on Machine Learning*. PMLR, 27496–27520.

[24] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th USENIX security symposium (USENIX Security 19)*. 729–746.

[25] Xingzhi Qian, Xinran Zheng, Yiling He, Shuo Yang, and Lorenzo Cavallaro. 2025. Lamd: Context-driven android malware detection and classification with llms. *arXiv preprint arXiv:2502.13055* (2025).

[26] Jordan Samhi and Alexandre Bartel. 2021. On the (in) effectiveness of static logic bomb detection for android apps. *IEEE Transactions on Dependable and Secure Computing* 19, 6 (2021), 3822–3836.

[27] Jordan Samhi, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. 2021. Raicc: Revealing atypical inter-component communication in android apps. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1398–1409.

[28] Jordan Samhi, Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2022. Difuzer: Uncovering suspicious hidden sensitive operations in android apps. In *Proceedings of the 44th International Conference on Software Engineering*. 723–735.

[29] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105.

[30] Tiezhu Sun, Nadia Daoudi, Kevin Allix, and Tegawendé F Bissyandé. 2021. Android malware detection: looking beyond dalvik bytecode. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 34–39.

[31] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. 2023. Automatic code summarization via chatgpt: How far are we? *arXiv preprint arXiv:2305.12865* (2023).

[32] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[33] VirusTotal. [n. d.]. VirusTotal: Free Online Virus, Malware and URL Scanner. https://www.virustotal.com/. Accessed: 2025-05-15.

[34] Wenxiang Zhao, Juntao Wu, and Zhaoyi Meng. 2025. Apppoet: Large language model based android malware detection via multi-view prompt engineering. *Expert Systems with Applications* 262 (2025), 125546.

[35] Li Zhong and Zilong Wang. 2024. Can llm replace stack overflow? a study on robustness and reliability of large language model code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 21841–21849.